APPENDIX A

XML
Extensible Markup Language
Introduction

Introduction

This document discusses some of the basic syntax and definitions pertaining to Extensible Markup Language (XML). This document will try to cover the types of scenarios encountered in the SCA Domain Profiles. Some rules of XML are discussed, along with document type definitions and their various components. First, however, a little history of markup languages will be covered.

Markup Languages

Databases categorize their data to allow them to perform tasks such as sorting, processing, and searching. To accomplish similar operations with other documents, the data would need to be "marked up". Markup is a way of dividing a continuous string of text (the document) into distinct pieces of data, thus adding context, or meaning, to raw text[1]. To apply markup to any document, codes (referred to as tags) are inserted.

Document Text – Not Tagged	Document Text - Tagged
Babe Ruth 714 Home Runs Bats Left Handed Elected to Hall of Fame 1936	<PlayerInfo> <Name> Babe Ruth </Name> <HomeRuns> 714 </Homeruns> <Bats> Left </Bats> <HallOfFame> 1936 </HallOfFame> </PlayerInfo>

To understand where XML fits in as a markup language, let's compare SGML, HTML, and XML.

SGML

Standard Generalized Markup Language (SGML) is the mother of all markup languages. Both HTML and XML are derived from SGML[2]. SGML defines a basic syntax, but allows the creation of unique elements (hence the term 'generalized'). To use SGML to describe a particular document, an appropriate set of elements and a document structure must be invented. A unique SGML application can be defined to describe a specific type of working document, or a standards body can define an SGML application to describe a commonly used document type. One famous example of this latter type of application is HTML, which is an SGML application developed in 1991 to describe Web pages. SGML might seem to be the perfect extensible language for describing Web documents. However, the W3C members who contemplate these issues deemed SGML too complicated and cumbersome to efficiently deliver information on the Web.

[1] "SoftQuad Xmetal 2.0 Courseware" (SoftQuad Software, Ltd.) p3
[2] "XML Step By Step" (Microsoft Press) Micheal J. Young p11

HTML
HyperText Markup Language (HTML) is an application of SGML. It has a predefined set of tags and does not offer the ability to define new elements or attributes. Even though HTML documents contain structural markup, most of the elements deal with generic structure and formatting and even though HTML elements do allow the creation of a hierarchy within a document, most documents use HTML non-hierarchically. The bottom line is that HTML is considered too flat, too rendering oriented, and could not easily support sophisticated processing.

XML
The eXtensible Markup Language (XML) is the result of a desire to create and implement a next-generation HTML. It is designed to structure data so that it can be easily transferred over a network and be consistently processed by the receiver. Because XML is used to describe information as well as structure it, it can be thought of as a data description language. XML can be used to describe data components, records and other data structures – even complex data structures.

Like HTML, XML is derived from SGML, but XML is actually a subset of the markup language and includes many of SGML's capabilities. With XML, data structures can be defined for documents using generalized markup – think of HTML tags and attributes but with more control. Unlike HTML, XML allows the definition of user-specified tags. This single feature frees one from the constraints of predefined tags and allows control in how the data is structured in an XML document.

Why Use XML?
With XML, there is total control over the structure, features, and content of documents. Furthermore, context or meaning is attached directly to content because each element can be customized. Elements aren't display oriented; they are information oriented. Finally, there is no proprietary format; the document or file is simply plain text with a .xml extension.

XML Architecture

This section examines XML architecture and how it relates to the structure. XML provides a three-part alternative to HTML.

- Structured Document: A structured document is marked up by an eXtensible Markup Language (XML). It is a document where content is placed where desired.
- Rules Definitions: Data schema file for describing rules for a particular type of document structure. Rules are contained in a Document Type Definition (DTD) file. This file defines elements, attributes, and the structure of the document.
- Style Sheet: A separate specification of rendering styles in a Cascading Style Sheet (CSS) or eXtensible StyleSheet Language (XLS) file. This file controls how the XML document displays on screen, such as in a browser or in an application.

An XML document is a structured document. XML requires a Rules Document (DTD or Schema) and allows for an optional Style Sheet file (CSS or XSL). The advantage of having a separate content document, structure document, and style document is that it allows tighter control over each of these key components. For example, assume that the style of the <Name> element needs to be modified from normal to bold letters for all XML documents; this can be done by changing one line in the style sheet file without having to modify each document, one by one. The following sections examine the structured document and the DTD. Style Sheets, however, will not be discussed further since the applications we develop for JTRS have no current need for them.

XML Structured Document

Besides content, the XML document contains markup comprised of three main components: elements, attributes and entities. The following sections introduce these components.

Element

Element is a structural component that describes content. Elements can contain content or they can be empty.

Elements with Content

Most elements have three parts: start tag, content, and end tag.

For example,

<Name> Babe Ruth </Name>

'<Name>' is the start tag.
'Babe Ruth' is the content.
'</Name>' is the end tag.

Empty Element

An empty element (that is, one without content) can also be entered into the document. An empty element is created by placing the end-tag immediately after the start-tag, or by using the special empty-element tag. These two notations have the same meaning:

<HR></HR> or <HR/>

The usefulness of an empty element may be questioned since it has no content. Here are two possible uses:

- An empty element can be used to tell the XML application to perform an action or display an object. In other words, the mere presence of an element with a particular name-without any content-can provide important information to the application.
- An empty element can store information through attributes (discussed later). This is the most common use of an empty element.

Legal Element Names

Element names must begin with a Unicode[3] letter, a colon (:) or an underscore (_). Other following characters in XML element names may be Unicode letters, period, hyphen, underscore, colon or digits. There is effectively no restriction on the lengths of characters in XML element names.

Well-Formed Documents

A well-formed document is one that conforms to the minimal set of rules that allow the document to be processed by a browser or other program. A well-formed XML document conforms to these rules:

- The document must have exactly one top-level element (the document element or root element). All other elements must be nested within it.

- Elements must be properly nested. That is, if an element starts within another element, it must also end within that same element.

- Each element must have both a start-tag and an end-tag. Unlike HTML, XML prohibits the omission of the end-tag – not even in situations where the browser would be able to figure out where the element ends. There is, however, a shortcut notation that can be used for an empty element – that is, an element with no content.

- The element-type name in a start-tag must exactly match the name in the corresponding end-tag.

- Element-type names are case sensitive. In fact, all text within XML markup is case sensitive. For example, the following element is illegal because the type name in the start-tag doesn't match the type name in the end-tag:

  <TITLE>Leaves of Grass</Title> <!--illegal element -->

[3] See http://www.unicode.org for more details on Unicode Character Set

Every XML document must be *well-formed*, meaning that it must meet the minimal requirements for a complying XML document. If a document isn't well-formed, it can't be considered an XML document.

Valid Documents
A well-formed XML document can also be *valid*. A valid XML document is a well-formed document that meets two further requirements:

- The prolog of the document must include a proper *document type declaration*, which contains a *document type definition* (DTD) that defines the structure of the document.
- The rest of the document must conform to the structure defined in the DTD.

Document Type Definitions (DTD) will be discussed in an upcoming section.

Attributes

Some XML elements have attributes. Attributes contain information intended for the application and for extra information associated with an element (like and ID number) used only by programs that read and write the file. For example,

```
<GREETING LANGUAGE="ENGLISH">
   Hello XML!
   <MOVIE DIRECTOR="Steven Spielberg"/>
</GREETING>
```

In the preceding example, the GREETING element has a LANGUAGE attribute, which has the value ENGLISH. The MOVIE element has a DIRECTOR attribute, which has the value "Steven Spielberg".

Elements can possess more than one attribute. However, end tags cannot possess attributes. The following example is illegal:

<SCRIPT>…</SCRIPT LANGUAGE="javascript" ENCODING="8859_1">

Another example, if "Name" is an empty element and it has "FirstName" and "LastName" as attributes, its declaration should be:

<Name FirstName="James" LastName="Bond"/>

Summary of Attribute Rules:
The following are a few of the basic rules for attribute declaration:

- In XML, all attribute values must be quoted.
- The attribute specifications appear in an element's start tag.
- There must be a space between the element name and the first attribute specification and between each attribute specification if there is more than one.

Entities

In the XML environment, an entity means a physical storage unit. Most of the time, this physical storage unit is just a file on a disc. Entities provide a solution to the problem of character set incompatibilities. An entity, properly defined, is simply a named unit of text that, when processed, is replaced by something else. They can be used to store frequently used blocks of text or to incorporate non-XML data into the document. Entities will not be discussed any further since the applications developed for JTRS have no need for them.

XML Schema

The purpose of an XML schema is to define a class of XML documents; thus, the term "instance document" is often used to describe an XML document that conforms to a particular schema. An XML Schema offers several features (such as a sophisticated set of basic data types) that are required for data processing applications. The SCA uses DTDs to define document structure, so XML Schema will not be discussed any further.

Document Type Definitions and Validity

Document Type Definitions (DTD) - A DTD provides a list of the elements, attributes, notations, and entities contained in a document, as well as their relationship to one another. DTD's specify a set of rules for the structure of a document. The DTD defines exactly what is and is not allowed to appear inside a document. It ensures the authors will use certain required elements, enter them in correct order, and supply the correct data.

Document Type Declarations – Document Type Declaration - Specifies the DTD a document uses. The document type declaration appears in a document's prolog, after the XML declaration but before the root element. It may contain the document type definition or a URL identifying the file where the DTD is found.

XML Prolog – The XML prolog is the first thing a processor – or human – sees in an XML document. It is placed at the top of the XML document, and it describes the document's content and structure. An XML prolog may include the following items:

- XML declaration
- Document type declaration
- Comments
- Processing Instructions
- White space

An XML prolog doesn't have to include any of that information. However, if a DTD is to be used, a few items must be included in the prolog in addition to an XML declaration. Here's an example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Book-Review SYSTEM "Book-Review.dtd">
<!--  End of Prolog -->
<!--  Beginning of Document Body  -->
<Book-Review>
…
<Book-Review>
<!-- End of Document Body -->
```

Included in this *prolog* is:

- the XML declaration itself
- the second line is a document type declaration
- the last lines are comments signifying the end of the prolog and the beginning of the document

In the previous example, the prolog pointed us to the location of the DTD (Book-Review.dtd). Now let's look at a DTD included in the prolog of an XML document:

```
<?xml version="1.0" standalone="yes"?>

<!-- File Name: hello.xml -->

<!DOCTYPE GREETING
[
  <!ELEMENT GREETING (#PCDATA)>
]>

<GREETING>
Hello XML!
</GREETING>
```

The *prolog* of this example document consists of three lines:

<?xml version="1.0" standalone="yes"?>

is the *XML Declaration*; which states that this is an XML document and gives the version number (currently at version 1.0). The XML declaration is optional, although the specification states that it should be included. *If it is not included, it must appear at the beginning of the document*.

The second line of the prolog consists of white space. Any amount of white space can be inserted between items in the prolog. The XML processor ignores it, but white space greatly enhances the readability.

The third line of the prolog is a comment. A *comment* (optional) begins with the <!-- characters and ends with the --> characters.

A document type declaration begins with <!DOCTYPE and ends with]>. It's customary to place the beginning and end on separate lines.

```
<!DOCTYPE GREETING
[
  <!ELEMENT GREETING (#PCDATA)>
]>
```

The name of the root element – GREETING in this example, follows <!DOCTYPE. This is not just a name, but a requirement. Any valid document with this document type declaration must have the root element GREETING.

```
<GREETING>
Hello XML!
</GREETING>
```

The content within the [ ] is the document type definition (DTD); in this case only one line, <!ELEMENT GREETING (#PCDATA)>

The single line <!ELEMENT GREETING (#PCDATA)> (case sensitive as most things are in XML) is an element type declaration. In this case, the name of the declared element is GREETING. It is the only element. This element may contain parsed character data.

Previous valid examples included the DTD in the document's prolog. The real power of XML, however, comes from common DTDs that can be shared among many documents written by different people. If the DTD is not directly included in the document but is linked in from an external source, changes made to the DTD automatically propagate to all documents using that DTD.

When an external DTD is used, the document type declaration changes. Instead of including the DTD in square brackets, the SYSTEM keyword is followed by an absolute or relative URL where the DTD can be found. For example:

<!DOCTYPE root_element_name SYSTEM "DTD_URL">

Here, root_element_name is simply the name of the root element as before, SYSTEM is an XML keyword, and DTD_URL is a relative or an absolute URL where the DTD can be found. For example:

<!DOCTYPE SEASON SYSTEM "baseball.dtd">

To convert a listing that uses an internal DTD to an external DTD, we first strip out the DTD and put it into its own file. This is everything between the opening <!DOCTYPE SEASON [and the closing]> exclusive. <!DOCTYPE SEASON [and]> are not included. This can be saved in a file called baseball.dtd. The filename is not important, though the extension .dtd is convention.

Next, the document itself must be modified. The XML declaration is no longer a standalone document because it depends on a DTD in another file. Therefore, the *standalone* attribute must be changed to *no*, as follows:

<?xml version="1.0" standalone="no"?>

Then, the <!DOCTYPE> tag must be changed so it points to the DTD by including the SYSTEM keyword and a URL (usually relative) where the DTD is found:

<!DOCTYPE SEASON SYSTEM "baseball.dtd">

The rest of the document is the same as before. However, now the prolog contains only the XML declaration and the document type declaration. It does not contain the DTD.

A document can't have more than one document type declaration, that is, more than one <!DOCTYPE> tag. To use elements declared in more than one DTD, external parameter entity references must be used.

DTDs designed for writers outside the creating organization use the *PUBLIC* keyword instead of the *SYSTEM* keyword. Furthermore, the DTD gets a name. The syntax follows:

<!DOCTYPE root_element_name PUBLIC "DTD_name" "DTD_URL">

If a DTD is an ISO standard, its name begins with the string "ISO." If a non-ISO standards body has approved the DTD, its name begins with a plus sign (+). If no standards body has approved the DTD, its name begins with a hyphen (-). These initial strings are followed by a double slash (//) and the name of the DTD's owner, which is followed by another double slash and the type of document the DTD describes. Then there's another double slash followed by an ISO 639 language identifier, such as EN for English. For example, the baseball DTD can be named as follows:

-//Elliotte Rusty Harold//DTD baseball statistics//EN

This example says this DTD is not standards-body approved (-), belongs to Elliotte Rusty Harold, describes baseball statistics, and is written in English. A full document type declaration pointing to this DTD with this name follows:

```
<!DOCTYPE SEASON PUBLIC
   "-//Elliotte Rusty Harold//DTD baseball statistics//EN"
   "http://metalab.unc.edu/xml/dtds/baseball.dtd"
```

If a particular document has a different structure than other pages on the site, it can be useful to define its structure in the document itself rather than in a separate DTD. This approach also makes the document easier to read. To this end, a document can use both an internal and an external DTD. Then internal declarations go inside square brackets at the end of the <!DOCTYPE> tag. In the event of a conflict between elements of the same name in the internal and external DTD subsets, the elements declared internally take precedence.

Building XML Documents

Here we take the contents discussed in prior sections and explain how to put the particulars to work. Examples will be provided showing more detailed usage of elements and attributes.

Declaring Element Types

In a valid XML document, explicit type declaration must be specified for every element used in the document – This is accomplished in an *element type declaration* within the DTD. An element declaration specifies the name and possible contents of an element. The list of contents is sometimes called the content specification. The content specification uses simple grammar to precisely specify what is and isn't allowed in a document. Below are a number of different types used to declare an element.

ANY

The first thing done in a document is to identify the root element. The !DOCTYPE declaration specifies this. In the example below, SEASON is the root element.

```
<!DOCTYPE SEASON
[
]>
```

However, this merely says that the root tag is SEASON. It doesn't say anything about what a SEASON element may or may not contain, which is why the SEASON element must be declared next in an element declaration. That's done with this line of code:

<!ELEMENT SEASON ANY>

All element type declarations begin with <!ELEMENT (case sensitive) and end with >. They include the name of the element being declared (SEASON in this example) followed by the content specification. The ANY keyword says that all possible elements as well as parsed character data can be children of the SEASON element.

#PCDATA

The keyword PCDATA stands for parsed character data. The XML processor parses character data within an element – that is, it scans the element looking for XML markup. Therefore, the right angle bracket (<) or ampersand (&) or the string]]> cannot be inserted as part of the character data because the parser would interpret each of these characters as markup. However, any other characters, including character references (such as '&#65' which represents a capital 'A') can be inserted.

Although any element may appear inside the document, elements that do appear must also be declared. The first one needed is YEAR. This is the element declaration for the YEAR element:

<!ELEMENT YEAR (#PCDATA)>

The SEASON and YEAR element declarations are included in the document type declaration, like this:

```
<!DOCTYPE SEASON
[
  <!ELEMENT SEASON ANY>
  <!ELEMENT YEAR (#PCDATA)>
]>
```

Sequence – A list of child elements separated by commas. In this declaration…

<!ELEMENT SEASON (YEAR, LEAGUE, LEAGUE)>

every valid SEASON element must contain exactly one YEAR element, followed by exactly two LEAGUE elements and nothing else. The complete document type declaration looks like this:

```
<!DOCTYPE SEASON
[
  <!ELEMENT YEAR (#PCDATA)>
  <!ELEMENT LEAGUE (LEAGUE_NAME)>
  <!ELEMENT LEAGUE_NAME (#PCDATA)>
  <!ELEMENT SEASON (YEAR, LEAGUE, LEAGUE)>
]>
```

How does one say that they want between four and six inclusive elements? Well, XML doesn't provide an easy way to do this. But it is possible to say *one or more of a given element is desired* by placing a plus sign (+) after the element name in the child list. For example:

<!ELEMENT DIVISION (DIVISION_NAME, TEAM+)>

This says that a DIVISION element must contain a DIVISION_NAME element followed by one or more TEAM elements.

If we want to specify that a TEAM can contain zero or more PLAYER children, we append an asterisk (*) to the element name in the child list. For example:

```
<!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER*)>
<!ELEMENT TEAM_CITY (#PCDATA)>
<!ELEMENT TEAM_NAME (#PCDATA)>
```

To indicate that we want zero or one element of the given type, we append a question mark (?) to the element.

The table below summarizes how ?, +, and * can be used with element declarations.

?	Zero or one of the preceding item
+	One or more of the preceding item
*	Zero or more of the preceding item

In general, a single parent element has many children. To indicate that the children must occur in sequence, they are separated by commas. However, each such child element may be suffixed with a question mark, a plus sign, or an asterisk to adjust the number of times it appears in that place in the sequence.

To indicate that the document author needs to input either one or another element, child elements are separated with a vertical bar (|) rather than a comma (,) in the parent's element declaration.

For example, the following says that the PAYMENT element must have a single child of type CASH, CREDIT_CARD, or CHECK

<!ELEMENT PAYMENT (CASH | CREDIT_CARD | CHECK)>

Mixed Content – When tags are declared that contain both child elements and parsed character data. This can be used to allow an arbitrary block of text to be suffixed after each TEAM. For example:

<!ELEMENT TEAM (#PCDATA | TEAM_CITY | TEAM_NAME | PLAYER)*>

The primary reason to mix content is during the process of converting old text data to XML, and testing the DTD by validation as new tags are added, rather than finishing the entire conversion and then trying to find the bugs. When the DTD is finished, it should not mix element children with parsed character data. A new tag can always be created that holds parsed character data. For example, a block of text can be included at the end of each TEAM element by declaring a new BLURB that holds only #PCDATA and adding it as the last child element of TEAM. Here's how this looks:

```
<!ELEMENT TEAM (TEAM_CITY, TEAM_PLAYER, PLAYER*, BLURB)>
<!ELEMENT BLURB (#PCDATA)>
```

In XML, empty elements are identified by empty tags that end with />, such as <IMG/>, <HR/>, and
.

Recall that valid documents must declare both the empty and nonempty elements used. Because empty elements by definition don't have children, they're easy to declare. Use an <!ELEMENT> declaration containing the name of the empty element as normal, but use the keyword EMPTY instead of a list of children. For example:

```
<!ELEMENT BR EMPTY>
<!ELEMENT IMG EMPTY>
<!ELEMENT HR EMPTY>
```

DTDs can contain comments, just like the rest of an XML document. These comments cannot appear inside a declaration, but they can appear outside one. For example, the element declaration for the YEAR element might have a comment such as this: XML processors will ignore comments.

```
<!-- Four-digit years like 1998, 1999, or 2000 -->
<!ELEMENT YEAR (#PCDATA)>
```

Attribute Declarations in DTDs

Like elements and entities, the attributes used in a document must be declared in the DTD for the document to be valid. The <!ATTLIST> tag declares attributes.<!ATTLIST> has the following form:

<!ATTLIST Element_name Attribute_name Type Default_value>

Element_name is the name of the element possessing this attribute.
Attribute_name is the name of the attribute.
Type is the kind of attribute – one of the ten valid types listed in the table below.
Default_value is the value the attribute takes on if no value is specified for the attribute.

Type	**Meaning**
CDATA	Character data – text that is not markup
Enumerated	A list of possible values from which exactly one will be chosen
ID	A unique name not shared by any other ID type attribute in the document
IDREF	The value of an ID type attribute of an element in the document
IDREFS	Multiple Ids of elements separated by white space
ENTITY	The name of an entity declared in the DTD
ENTITIES	The names of multiple entities declared in the DTD, separated by white space
NMTOKEN	An XML name
NMTOKENS	Multiple XML names separated by white space
NOTATION	The name of a notation declared in the DTD

For example, consider the following element:

```
<GREETING LANGUAGE=”Spanish”>
   Hola!
</GREETING>
```

This element might be declared as follows in the DTD:

```
<!ELEMENT GREETING (#PCDATA)>
<!ATTLIST GREETING LANGUAGE CDATA “English”>
```

The <!ELEMENT> tag simply says that a greeting element contains parsed character data. That's nothing new. The <!ATTLIST> tag says that GREETING elements have an attribute with the name LANGUAGE whose value has the type CDATA-which is essentially the same as #PCDATA for element content. If a GREETING tag is encountered without a LANGUAGE attribute, the value English is used by default.

The attribute list is declared separately from the tag itself. The name of the element to which the attribute belongs is included in the <!ATTLIST> tag. This attribute declaration applies only to that element, which is GREETING in the preceding example. If other elements also have LANGUAGE attributes, they require separate <!ATTLIST> declarations.

Declaring Multiple Attributes

Elements often have multiple attributes. XML tags can also have multiple attributes. For instance, a RECTANGLE element naturally needs both a LENGTH and a WIDTH.

```
<RECTANGLE LENGTH=”70px” WIDTH=”85px”/>
```

These attributes can be declared in several attribute declarations, with one declaration for each attribute. For example:

```
<!ELEMENT RECTANGLE EMPTY>
<!ATTLIST RECTANGLE LENGTH CDATA “0px”>
<!ATTLIST RECTANGLE WIDTH    CDATA “0px >
```

The two <!ATTLIST> tags can be combined into one: (*not recommended, however*)

```
<!ATTLIST RECTANGLE  LENGTH CDATA “0px”
                     WIDTH    CDATA “0px >
```

Specifying Default Values for Attributes

Instead of specifying an explicit default attribute value like 0px, an attribute declaration can require the author to provide a value, allow the value to be omitted completely, or even always use the default value. These requirements are specified with the three keywords #REQUIRED, #IMPLIED, and #FIXED, respectively.

#REQUIRED – While XML can't prevent someone from attributing authorship to "Luke Skywalker", it can at least require that authorship is attributed to someone by using #REQUIRED as the default value. For example:

```
<!ELEMENT  AUTHOR  EMPTY>
<!ATTLIST  AUTHOR  NAME CDATA  #REQUIRED>
<!ATTLIST  AUTHOR  EMAIL  CDATA  #REQUIRED>
<!ATTLIST  AUTHOR  EXTENSION  CDATA  #REQUIRED>
```

If the parser encounters an <AUTHOR/> tag that does not include one or more of these attributes, it returns an error.

#IMPLIED – Sometimes a good option for a default value may not exist, but it also may not be desirable to require the author of the document to include a value. In other words, the field is optional. For example:

```
<!ELEMENT  AUTHOR  EMPTY>
<!ATTLIST  AUTHOR  NAME CDATA  #REQUIRED>
<!ATTLIST  AUTHOR  EMAIL  CDATA  #REQUIRED>
<!ATTLIST  AUTHOR  EXTENSION  CDATA  #IMPLIED>
```

#FIXED – It may be desired to provide a default value for the attribute without allowing the author to change it. For example, it may be required that everyone uses a specified value for the COMPANY attribute:

```
<!ELEMENT  AUTHOR  EMPTY>
<!ATTLIST  AUTHOR  NAME         CDATA  #REQUIRED>
<!ATTLIST  AUTHOR  EMAIL        CDATA  #REQUIRED>
<!ATTLIST  AUTHOR  EXTENSION  CDATA  #IMPLIED>
<!ATTLIST  AUTHOR  COMPANY    CDATA  #FIXED  "TIC">
```

Document authors are not required to actually include the fixed attribute in their tags. If they don't include the fixed attribute, the default value will be used. If they do, however, they must use an identical value. Otherwise, the parser will return an error.

Specifying an Enumerated Type

Like all attributes, the value assigned to an enumerated type must be a quoted string, and its value must match one of the names listed in the attribute-type specification, which can have one of two forms. However, the form that uses the NOTATION keyword will not be discussed in this paper.

The other form requires an open parenthesis, followed by a list of name tokens separated with | characters, followed by a close parenthesis. For example, to restrict the values of the *Class* attribute to "action", "comedy", or "musical", the attribute could be defined as an enumerated type, like this:

```
<!ATTLIST FILM
   Class (action | comedy | musical) "comedy">
```

Here's a complete XML document that shows the used of the *Class* attribute:

```
<?xml version="1.0"?>

<!DOCTYPE  FILM
[
  <!ELEMENT FILM (TITLE, (STAR | NARRATOR | INSTRUCTOR))>

   <!ATTLIST FILM
      Class (action | comedy | musical) "comedy">

   <!ELEMENT TITLE (#PCDATA)>
   <!ELEMENT STAR (#PCDATA)>
   <!ELEMENT NARRATOR (#PCDATA)>
   <!ELEMENT INSTUCTOR (#PCDATA)>
]

<FILM Class="action">

   <TITLE> The Terminator </TITLE>
   <STAR> Arnold Schwarzenegger </STAR>

</FILM>
```

Attributes versus Elements

There are no hard and fast rules about when to use child elements and when to use attributes. One good rule of thumb is that the data itself should be stored in elements. Information about the data (meta-data) should be stored in attributes. And when in doubt, put the information in elements.

To differentiate between data and meta-data, one needs to ask whether someone reading the document would want to see a particular piece of information. If the answer is yes, then the information probably belongs in a child element. If the answer is no, then the information probably belongs in an attribute. Attributes are good places to put ID numbers, URLs, references, and other information not directly or immediately relevant to the reader. However, there are many exceptions to the basic principal of storing meta-data as attributes. These include:

- Attributes can't hold structure well
- Elements allow the inclusion of meta-meta-data (information about the information about the information).
- Not everyone agrees on what is and isn't meta-data.
- Elements are more extensible in the face of future changes.

One important principal to remember is that elements can have substructure and attributes can't. This makes elements far more flexible, and may be a convincing enough reason to encode meta-data as child elements.

Furthermore, using elements instead of attributes makes it straightforward to include additional information like the author's e-mail address, a URL where an electronic copy of the document can be found, or anything else that seems important.

Dates are a common example. One common piece of meta-data about scholarly articles is the date the article was first received. This is important for establishing priority of discovery and invention. It's easy to include a DATE attribute in an ARTICLE tag like this:

```
<ARTICLE  DATE = "06/14/1969">
  Birthday of Tennis Great Steffi Graf
</ARTICLE>
```

However, the DATE attribute has substructure signified by the /. Getting that structure out of the attribute value, however, is much more difficult than reading child elements of a DATE element, as shown below:

```
<DATE>
  <YEAR> 1969 </YEAR>
  <MONTH> 06 </MONTH>
  <DAY> 14 </DAY>
</DATE>
```